Parallel Computing for Science & Engineering

3/28/2023

Instructors:

Lars Koesterke, TACC

Victor Eijkhout, TACC

OpenMP Clauses

Clauses control the behavior of an OpenMP directive:

Control	Clause
Schedule for *for/do* worksharing	*schedule( )*
Data Scoping	*private( ), shared( ), default( )*
Initialization	*firstprivate( )*
Parallelize a region or not	*if( )*
Number of threads to use	*num_threads( )*

Schedule Clause for loop worksharing

schedule(static)

Each CPU receives one set of contiguous iterations

```
!$omp do schedule(...)
   do i=1,128
     A(i)=B(i)+C(i)
   enddo
```

schedule(static, C)

Iterations are divided round-robin fashion in chunks of size C

schedule(dynamic, C)

Iterations handed out in chunks of size C as CPUs become available

schedule(guided, C)

Each of the iterations are handed out in pieces of logarithmically decreasing size, with C minimum number of iterations to dispatch each time

schedule (runtime)

Schedule and chunk size taken from the OMP_SCHEDULE environment variable

Example - schedule(static,16), threads = 4

```
!$omp parallel do schedule(static,16)
      do i=1,128
       A(i)=B(i)+C(i)
      enddo
```

thread0:
`do i=1,16`
`  A(i)=B(i)+C(i)`
`enddo`
`do i=65,80`
`  A(i)=B(i)+C(i)`
`enddo`
	thread2:
	`do i=33,48`
	`  A(i)=B(i)+C(i)`
	`enddo`
	`do i = 97,112`
	`  A(i)=B(i)+C(i)`
	`enddo`

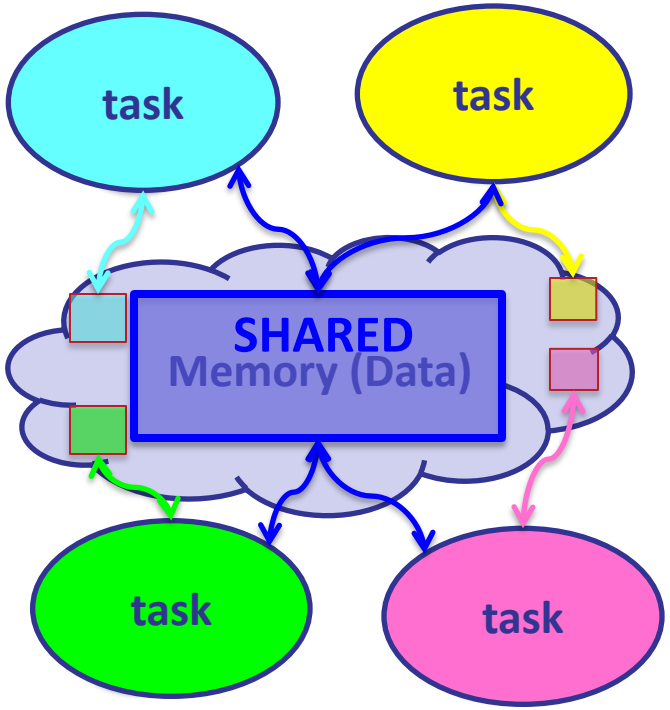
thread1:
`do i=17,32`
`  A(i)=B(i)+C(i)`
`enddo`
`do i = 81,96`
`  A(i)=B(i)+C(i)`
`enddo`
	thread3:
	`do i=49,64`
	`  A(i)=B(i)+C(i)`
	`enddo`
	`do i = 113,128`
	`  A(i)=B(i)+C(i)`
	`enddo`

Comparison of Scheduling Options

name	type	chunk	chunk size	chunk #	static or dynamic	compute overhead
partitioned	static	no	N/P	P	static	lowest
interleaved	static	yes	C	N/C	static	low
simple dynamic	dynamic	optional	C	N/C	dynamic	medium
guided	dynamic	optional	decreasing from N/P*	no fewer than N/C	dynamic	Less than medium
runtime	runtime	no	varies	varies	varies	varies

*Decreases as unassigned/P

Data Model – for parallel region



Private Memory

- Threads Execute on Cores/HW-threads
- In a parallel region, team threads are assigned (tied) to implicit tasks to do work. Think of tasks and threads as being synonymous.
- Tasks by "default" share memory declared in scope before a parallel region.
- Data: shared or private
  - Shared data: accessible by all tasks
  - Private data: only accessible by the owner task

OpenMP Data Environment

- Clauses control the data-sharing attributes of variables within a parallel region:

shared, private, reduction, firstprivate, lastprivate

Default variable scope (in parallel region):

1. Variables declared in main/program (C/F90) are shared by default
2. Global variables are shared by default
3. Automatic variables within subroutines called from within a parallel region are private (reside on a stack private to each thread)
4. Loop index of worksharing loops are private.
5. Default scoping rule can be changed with `default(none)` clause

C/C++

Private & Shared Data

shared - Variable is shared (seen) by all threads

private - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

```
!$omp parallel do shared(a,b,c,n)  private(i)
      do i = 1,n
            a(i) = b(i) + c(i)
      enddo
```

OK to be explicit;
but not necessary.

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i

F90

Private & Shared Data

shared - Variable is shared (seen) by all threads

private - Each thread has a private instance (copy) of the variable

Defaults: The for-loop index is private, all other variables are shared

```
#pragma omp parallel for shared(a,b,c,n)  private(i)
      for (i=0; i<n; i++){
            a[i] = b[i] + c[i];
         }
```

OK to be explicit;
but not necessary.

All threads have access to the same storage areas for a, b, c, and n, but each loop has its own private copy of the loop index, i

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
#pragma omp parallel for shared(a,b,c,n) private(temp,i)
for (i=0; i<n; i++){
    temp = a[i] / b[i];
    c[i] = temp + cos(temp);
}
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel for is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

F90

Private Data Example

- In the following loop, each thread needs its own private copy of temp
- If temp were shared, the result would be unpredictable since each thread would be writing/reading to/from the same memory location

```
!$omp parallel for shared(a,b,c,n) private(temp,i)
do i = 1,n
   temp = a(i) / b(i)
   c(i) = temp + cos(temp)
endo
```

- A **lastprivate(temp)** clause will copy the last loop(stack) value of temp to the (global) temp storage when the parallel DO is complete.
- A **firstprivate(temp)** would copy the global temp value to each stack's temp.

Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float asum=0.0, aprod=1.0;

#pragma omp parallel for reduction(+:asum) reduction(*:aprod)
  for (i=0; i<n; i++){
    asum  = asum  + a[i];
    aprod = aprod * a[i];
  }
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

F90

Reduction

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
real asum=0.0, aprod=1.0

!$omp parallel do reduction(+:asum) reduction(*:aprod)
   do i = 1,n
      asum  = asum  + a(i)
      aprod = aprod * a(i)
   enddo
print*, asum, aprod
```

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

- **After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction**

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High-Level Synchronization
  - critical
  - atomic
  - barrier
  - ordered
- Low-Level Synchronization
  - locks

C/C++

Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the #pragma omp atomic directive for simple cases: can use hardware support

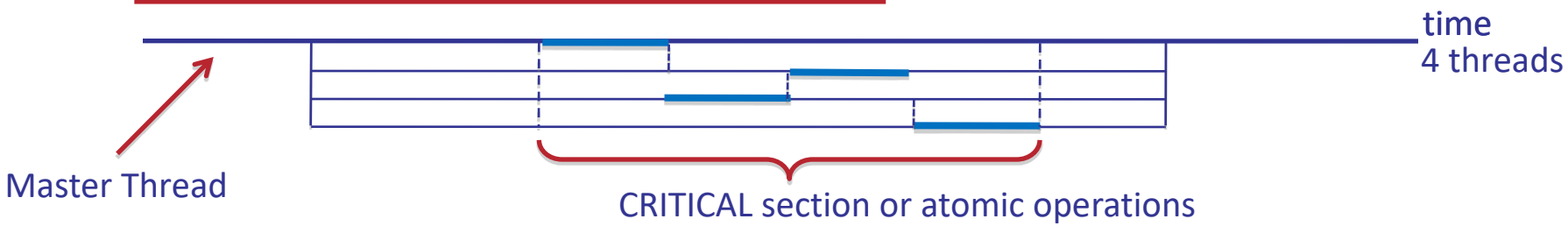
```
#pragma omp parallel shared(sum,x,y)
{...
   #pragma omp critical

      update(x);
      update(y);
      sum=sum+1;
   }
....
}
```

```
#pragma omp parallel shared(sum)
{...

   #pragma omp atomic
   sum=sum+1
...
}
```

Atomic has read, write, update, capture clauses.



F90

Synchronization: Critical/Atomic Directives

- When each thread must execute a section of code serially the region must be marked with critical/end critical directives
- Use the !$ omp atomic directive for simple cases: can use hardware support

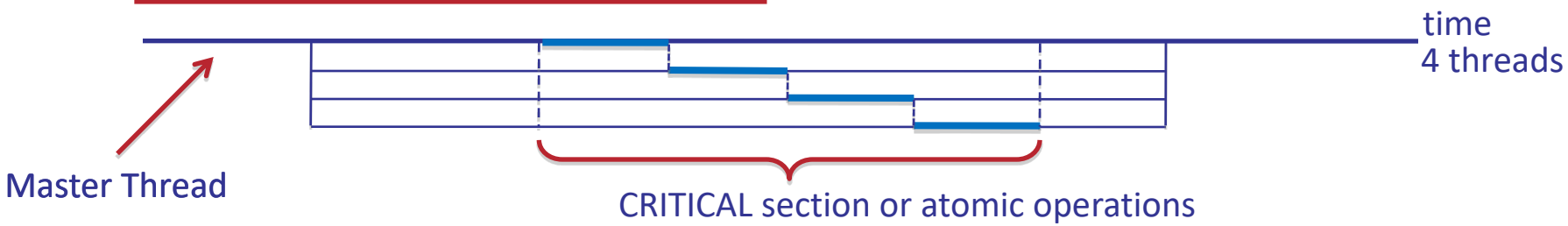
```
!$omp parallel shared(sum,x,y)
 ...

   !$omp critical
     update(x);
     update(y);
     sum=sum+1;
   !$omp end critical
....
!$omp end parallel
```

```
!$omp parallel shared(sum)
...

      !$omp atomic
      sum=sum+1;
...
!$omp end parallel
```

Atomic has read, write, update, capture clauses.



Single Construct

Single:

- Work-sharing construct
- Any single thread executes the construct.
- Implied barrier.

Fortran

```
!$omp parallel private(id)
  id=omp_get_thread_num()
  !$omp single
    x = 1
  !$omp end single
  call foo(id,x)
!$omp end parallel
```

C/C++

```
#pragma omp parallel private(id)
{
  id=omp_get_thread_num();
  #pragma omp single
  x = 1;

  foo(id,x);
}
```

Master Construct

Master
- Not a work-sharing construct
- Only the master executes the construct.
- No implied barrier

race condition

race condition

Fortran

```
!$omp parallel private(id)
  id=omp_get_thread_num()
  !$omp master
    x = 1
  !$omp end master
  call foo(id,x)
!$omp end parallel
```

C/C++

```
#pragma omp parallel private(id)
{
  id=omp_get_thread_num();
  #pragma omp master
  x = 1;

  foo(id,x);
}
```

Progression

Concept	What to learn	Level	Optimize
Setup	How to compile OMP_NUM_THREADS	Basic	1 thread per ‘core’
Parallel region	Forking/joining threads	Easy	Minimize number of fork/join
Work-sharing/replicated work	What do the threads do? ‘omp do/for’	Work-sharing: easy Replicated: medium	Optimize scheduling Remove implicit barriers Limit replicated/single work
Avoiding race conditions	----------	Will take effort!	----------
- Private variables	Why/how to shelter data	medium	
- reduction	Condensing a result from pieces	medium	
- Critical/atomic	All threads, but one thread at a time	harder	Do not serialize everything
- Single/master	One thread, and only one thread	harder	See ‘min. fork/join’
Advanced	----------		
- Hybrid	MPI + OpenMP	medium	Interplay MPI/OpenMP
- Thread/memory pinning		medium	Utilize all cores
- SIMD	Vectorization with OpenMP	hard	Utilize vector lanes
- Tasking	Irregular problems	hard	
Accelerators/GPUs	Many new concepts	‘Different’	

NOWAIT

- When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
- By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
#pragma omp parallel
{
  #pragma omp for nowait
       {
          for (i=0; i<n; i++)
            {work(i);}
       }
  #pragma omp for schedule(guided,k)
       {
          for (i=0; i<m; i++)
            {x[i]=y[i]+z[i];}
       }
}
```

F90

NOWAIT

- When a work-sharing region is exited, a barrier is implied - all threads must reach the barrier before any can proceed.
- By using the NOWAIT clause at the end of each loop inside the parallel region, an unnecessary synchronization of threads can be avoided.

```
!$OMP PARALLEL
   !$OMP DO
      do i=1,n
         work(i)
      enddo
   !$OMP END DO NOWAIT
   !$OMP DO schedule(guided,k)
      do i=1,m
         x(i)=y(i)+z(i)
      enddo
   !$OMP END DO
!$OMP END PARALLEL
```

Runtime Library Routines

function	description
omp_get_num_threads()	Number of threads in team, N
omp_get_thread_num()	Thread ID {0 -> N-1}
omp_get_num_procs()	Number of machine CPUs
omp_in_parallel()	True if in parallel region & multiple thread executing
omp_set_num_threads(#)	Set the number of threads in the team

Environment Variables

variable	description
OMP_NUM_THREADS=integer	Set to default no. of threads to use
OMP_SCHEDULE="schedule-type[, chunk_size]"	Sets "runtime" in loop schedule clause: "…omp for/do schedule(runtime)"
OMP_DISPLAY_ENV=anyvalue	Prints runtime environment at beginning of code execution.

[…] = optional

OpenMP Wallclock Timers

```
real*8 :: omp_get_wtime,   omp_get_wtick()        (Fortran)
double    omp_get_wtime(), omp_get_wtick();       (C)
```

```
double t0, t1, dt, res;
...
t0 = omp_get_wtime();
<work>
t1 = omp_get_wtime();
dt = t1 - t0;

res = 1.0/omp_get_wtick();
printf("Elapsed time = %lf\n",dt);
printf("clock resolution = %lf\n",res);
```

C/C++

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
#pragma omp parallel num_threads(scalar int expression)
{
    <code block>
}
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function

F90

NUM_THREADS clause

- Use the NUM_THREADS clause to specify the number of threads to execute a parallel region

```
!$omp parallel num_threads(scalar integer expression)
    <code block>
!$omp end parallel
```

where scalar integer expression must evaluate to a positive integer

- num_threads() supersedes the number of threads specified by the OMP_NUM_THREADS environment variable or that set by the OMP_SET_NUM_THREADS function